

Chapter B7. Random Numbers

One might think that good random number generators, including those in Volume 1, should last forever. The world of computing changes very rapidly, however:

- When Volume 1 was published, it was unusual, except on the fastest supercomputers, to “exhaust” a 32-bit random number generator, that is, to call for all 2^{32} sequential random values in its periodic sequence. Now, this is feasible, and not uncommon, on fast desktop workstations. A useful generator today must have a minimum of 64 bits of state space, and generally somewhat more.
- Before Fortran 90, the Fortran language had no standardized calling sequence for random numbers. Now, although there is still no standard *algorithm* defined by the language (rightly, we think), there is at least a standard calling sequence, exemplified in the intrinsics `random_number` and `random_seed`.
- The rise of parallel computing places new algorithmic demands on random generators. The classic algorithms, which compute each random value from the previous one, evidently need generalization to a parallel environment.
- New algorithms and techniques have been discovered, in some cases significantly faster than their predecessors.

These are the reasons that we have decided to implement, in Fortran 90, different uniform random number generators from those in Volume 1’s Fortran 77 implementations. We hasten to add that there is nothing wrong with any of the generators in Volume 1. That volume’s `ran0` and `ran1` routines are, to our knowledge, completely adequate as 32-bit generators; `ran2` has a 64-bit state space, and our previous offer of \$1000 for *any* demonstrated failure in the algorithm has never yet been claimed (see [1]).

Before we launch into the discussion of parallelizable generators with Fortran 90 calling conventions, we want to attend to the continuing needs of longtime “`x=ran(idum)`” users with purely serial machines. If you are a satisfied user of Volume 1’s `ran0`, `ran1`, or `ran2` Fortran 77 versions, you are in this group. The following routine, `ran`, preserves those routines’ calling conventions, is considerably faster than `ran2`, and does not suffer from the old `ran0` or `ran1`’s 32-bit period exhaustion limitation. It is completely portable to all Fortran 90 environments. We recommend `ran` as the plug-compatible replacement for the old `ran0`, `ran1`, and `ran2`, and we happily offer exactly the same \$1000 reward terms as were (and are still) offered on the old `ran2`.

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
<http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcserv@cambridge.org (outside North America).

```

FUNCTION ran(idum)
IMPLICIT NONE
INTEGER, PARAMETER :: K4B=selected_int_kind(9)
INTEGER(K4B), INTENT(INOUT) :: idum
REAL :: ran
  "Minimal" random number generator of Park and Miller combined with a Marsaglia shift
  sequence. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint
  values). This fully portable, scalar generator has the "traditional" (not Fortran 90) calling
  sequence with a random deviate as the returned function value: call with idum a negative
  integer to initialize; thereafter, do not alter idum except to reinitialize. The period of this
  generator is about  $3.1 \times 10^{18}$ .
  INTEGER(K4B), PARAMETER :: IA=16807, IM=2147483647, IQ=1277773, IR=2836
  REAL, SAVE :: am
  INTEGER(K4B), SAVE :: ix=-1, iy=-1, k
  if (idum <= 0 .or. iy < 0) then           Initialize.
    am=nearest(1.0, -1.0)/IM
    iy=ieor(ieor(888889999, abs(idum)), 1)
    ix=ieor(777755555, abs(idum))
    idum=abs(idum)+1                         Set idum positive.
  end if
  ix=ieor(ix, ishft(ix, 13))                Marsaglia shift sequence with period  $2^{32} - 1$ .
  ix=ieor(ix, ishft(ix, -17))
  ix=ieor(ix, ishft(ix, 5))
  k=iy/IQ                                     Park-Miller sequence by Schrage's method,
  iy=IA*(iy-k*IQ)-IR*k                      period  $2^{31} - 2$ .
  if (iy < 0) iy=iy+IM
  ran=am*ieor(iand(IM, ieor(ix, iy)), 1)     Combine the two generators with masking to
                                               ensure nonzero value.
END FUNCTION ran

```

This is a good place to discuss a new bit of algorithmics that has crept into `ran`, above, and even more strongly affects all of our new random number generators, below. Consider:

```

ix=ieor(ix, ishft(ix, 13))
ix=ieor(ix, ishft(ix, -17))
ix=ieor(ix, ishft(ix, 5))

```

These lines update a 32-bit integer `ix`, which cycles pseudo-randomly through a full period of $2^{32} - 1$ values (excluding zero) before repeating. Generators of this type have been extensively explored by Marsaglia (see [2]), who has kindly communicated some additional results to us in advance of publication. For convenience, we will refer to generators of this sort as "Marsaglia shift registers."

Useful properties of Marsaglia shift registers are (i) they are very fast on most machines, since they use only fast logical operations, and (ii) the bit-mixing that they induce is quite different in character from that induced by arithmetic operations such as are used in linear congruential generators (see Volume 1) or lagged Fibonacci generators (see below). Thus, the combination of a Marsaglia shift register with another, algorithmically quite different generator is a powerful way to suppress any residual correlations or other weaknesses in the other generator. Indeed, Marsaglia finds (and we concur) that the above generator (with constants 13, -17, 5, as shown) is *by itself* about as good as any 32-bit random generator.

Here is a very brief outline of the theory behind these generators: Consider the 32 bits of the integer as components in a vector of length 32, in a linear space where addition and multiplication are done modulo 2. Noting that exclusive-or (`ieor`) is the same as addition, each of the three lines in the updating can be written as the action of a 32×32 matrix on a vector, where the matrix is all zeros except for ones on

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

the diagonal, and on exactly one super- or subdiagonal (corresponding to positive or negative second arguments in `ishft`). Denote this matrix as \mathbf{S}_k , where k is the shift argument. Then, one full step of updating (three lines of code, above) corresponds to multiplication by the matrix $\mathbf{T} \equiv \mathbf{S}_{k_3}\mathbf{S}_{k_2}\mathbf{S}_{k_1}$.

One next needs to find triples of integers (k_1, k_2, k_3) , for example $(13, -17, 5)$, that give the full $M \equiv 2^{32} - 1$ period. Necessary and sufficient conditions are that $\mathbf{T}^M = \mathbf{1}$ (the identity matrix), and that $\mathbf{T}^N \neq \mathbf{1}$ for these five values of N : $N = 3 \times 5 \times 17 \times 257$, $N = 3 \times 5 \times 17 \times 65537$, $N = 3 \times 5 \times 257 \times 65537$, $N = 3 \times 17 \times 257 \times 65537$, $N = 5 \times 17 \times 257 \times 65537$. (Note that each of the five prime factors of M is omitted one at a time to get the five values of N .) The required large powers of \mathbf{T} are readily computed by successive squarings, requiring only on the order of $32^3 \log M$ operations. With this machinery, one can find full-period triples (k_1, k_2, k_3) by exhaustive search, at reasonable cost.

Not all such triples are equally good as generators of random integers, however. Marsaglia subjects candidate values to a battery of tests for randomness, and we have ourselves applied various tests. This stage of winnowing is as much art as science, because all 32-bit generators can be made to exhibit signs of failure due to period exhaustion (if for no other reason). “Good” triples, in order of our preference, are $(13, -17, 5)$, $(5, -13, 6)$, $(5, -9, 7)$, $(13, -17, 15)$, $(16, -7, 11)$. When a full-period triple is good, its reverse is also full-period, and also generally good. A good *quadruple* due to Marsaglia (generalizing the above in the obvious way) is $(-4, 8, -1, 5)$. We would not recommend relying on any single Marsaglia shift generator (nor on any other simple generator) *by itself*. Two or more generators, of quite different types, should be combined [1].

* * *

 Let us now discuss explicitly the needs of *parallel* random number generators. The general scheme, from the user’s perspective, is that of Fortran 90’s intrinsic `random_number`: A statement like `call ran1(harvest)` (where `ran1` will be one of our portable replacements for the compiler-dependent `random_number`) should fill the real array `harvest` with pseudo-random real values in the range $(0, 1)$. Of course, we want the underlying machinery to be completely parallel, that is, no do-loops of order $N \equiv \text{size}(\text{harvest})$.

A first design decision is whether to replicate the state-space across the parallel dimension N , i.e., whether to reserve storage for essentially N scalar generators. Although there are various schemes that avoid doing this (e.g., mapping a single, smaller, state space into N different output values on each call), we think that it is a memory cost well worth paying in return for achieving a less exotic (and thus better tested) algorithm. However, this choice dictates that we must keep the state space *per component* quite small. We have settled on five or fewer 32-bit words of state space per component as a reasonable limit. Some otherwise interesting and well tested methods (such as Knuth’s subtractive generator, implemented in Volume 1 as `ran3`) are ruled out by this constraint.

A second design decision is how to initialize the parallel state space, so that different parallel components produce different sequences, and so that there is an acceptable degree of randomness *across* the parallel dimension, as well as *between successive calls* of the generator. Each component starts its life with one and only one unique identifier, its component index n in the range $1 \dots N$. One is

tempted simply to hash the values n into the corresponding components of initial state space. “Random” hashing is a bad idea, however, because different n ’s will produce identical 32-bit hash results by chance when N is no larger than $\sim 2^{16}$. We therefore prefer to use a kind of reversible pseudo-encryption (similar to the routine `psdes` in Volume 1 and below) which guarantees causally that different n ’s produce different state space initializations.



The machinery for allocating, deallocating, and initializing the state space, including provision of a user interface for getting or putting the contents of the state space (as in the intrinsic `random_seed`) is fairly complicated. Rather than duplicate it in each different random generator that we provide, we have consolidated it in a single module, `ran_state`, whose contents we will now discuss. Such a discussion is necessarily technical, if not arcane; on first reading, you may wish to skip ahead to the actual new routines `ran0`, `ran1`, and `ran2`. If you do so, you will need to know only that `ran_state` provides each vector random routine with five 32-bit vectors of state information, denoted `iran`, `jran`, `kran`, `mran`, `nran`. (The overloaded scalar generators have five corresponding 32-bit scalars, denoted `iran0`, etc.)

`MODULE ran_state`

This module supports the random number routines `ran0`, `ran1`, `ran2`, and `ran3`. It provides each generator with five integers (for vector versions, five vectors of integers), for use as internal state space. The first three integers (`iran`, `jran`, `kran`) are maintained as nonnegative values, while the last two (`mran`, `nran`) have 32-bit nonzero values. Also provided by this module is support for initializing or reinitializing the state space to a desired standard sequence number, hashing the initial values to random values, and allocating and deallocating the internal workspace.

```
USE nrtype
IMPLICIT NONE
INTEGER, PARAMETER :: K4B=selected_int_kind(9)
Independent of the usual integer kind I4B, we need a kind value for (ideally) 32-bit integers.
INTEGER(K4B), PARAMETER :: hg=huge(1_K4B), hgm=-hg, hgng=hgm-1
INTEGER(K4B), SAVE :: lenran=0, seq=0
INTEGER(K4B), SAVE :: iran0,jran0,kran0,nran0,mran0,rans
INTEGER(K4B), DIMENSION(:, :, ), POINTER, SAVE :: ranseeds
INTEGER(K4B), DIMENSION(:, ), POINTER, SAVE :: iran,jran,kran, &
nran,mran,rans
REAL(SP), SAVE :: amm
INTERFACE ran_hash
  MODULE PROCEDURE ran_hash_s, ran_hash_v
END INTERFACE
CONTAINS
```

Scalar and vector versions of the hashing procedure.

(We here intersperse discussion with the listing of the module.) The module defines `K4B` as an integer `KIND` that is intended to be 32 bits. If your machine doesn’t have 32-bit integers (hard to believe!) this will be caught later, and an error message generated. The definition of the parameters `hg`, `hgm`, and `hgng` makes an assumption about 32-bit integers that goes beyond the strict Fortran 90 integer model, that the magnitude of the most negative representable integer is greater by one than that of the most positive representable integer. This is a property of the *two’s complement arithmetic* that is used on virtually all modern machines (see, e.g., [3]).

The global variables `rans` (for scalar) and `ravn` (for vector) are used by all of our routines to store the *integer* value associated with the most recently returned call. You can access these (with a “`USE ran_state`” statement) if you want integer, rather than real, random deviates.

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

The first routine, `ran_init`, is called by routines later in the chapter to initialize their state space. It is *not* intended to be called from a user's program.

```
SUBROUTINE ran_init(length)
USE nrtype; USE nrutil, ONLY : arth,nrerror,reallocate
IMPLICIT NONE
INTEGER(K4B), INTENT(IN) :: length
Initialize or reinitialize the random generator state space to vectors of size length. The
saved variable seq is hashed (via calls to the module routine ran_hash) to create unique
starting seeds, different for each vector component.
INTEGER(K4B) :: new,j,hgt
if (length < lenran) RETURN
Simply return if enough space is already al-
hgt=hg
located.
The following lines check that kind value K4B is in fact a 32-bit integer with the usual properties
that we expect it to have (under negation and wrap-around addition). If all of these tests are
satisfied, then the routines that use this module are portable, even though they go beyond
Fortran 90's integer model.
if (hg /= 2147483647) call nrerror('ran_init: arith assump 1 fails')
if (hgng >= 0) call nrerror('ran_init: arith assump 2 fails')
if (hgt+1 /= hgng) call nrerror('ran_init: arith assump 3 fails')
if (not(hg) >= 0) call nrerror('ran_init: arith assump 4 fails')
if (not(hgng) < 0) call nrerror('ran_init: arith assump 5 fails')
if (hg+hgng >= 0) call nrerror('ran_init: arith assump 6 fails')
if (not(-1_k4b) < 0) call nrerror('ran_init: arith assump 7 fails')
if (not(0_k4b) >= 0) call nrerror('ran_init: arith assump 8 fails')
if (not(1_k4b) >= 0) call nrerror('ran_init: arith assump 9 fails')
if (lenran > 0) then
    ranseeds=>realloc(reallocate(ranseeds,length,5))
    ranv=>realloc(reallocate(ranv,length-1))
    new=lenran+1
else
    allocate(ranseeds(length,5))
    allocate(ranv(length-1))
    new=1
    amm=nearest(1.0_sp,-1.0_sp)/hgng
    Index of first location not yet initialized.
    Use of nearest is to ensure that returned random deviates are strictly less than 1.0.
    if (amm*hgng >= 1.0 .or. amm*hgng <= 0.0) &
        call nrerror('ran_init: arith assump 10 fails')
end if
Set starting values, unique by seq and vector component.
ranseeds(new:,1)=seq
ranseeds(new:,2:5)=spread(arth(new,1,size(ranseeds(new:,1))),2,4)
do j=1,4
    Hash them.
    call ran_hash(ranseeds(new:,j),ranseeds(new:,j+1))
end do
where (ranseeds(new:,1:3) < 0) &
    Enforce nonnegativity.
    ranseeds(new:,1:3)=not(ranseeds(new:,1:3))
where (ranseeds(new:,4:5) == 0) ranseeds(new:,4:5)=1
    Enforce nonzero.
if (new == 1) then
    Set scalar seeds.
    iran0=ranseeds(1,1)
    jran0=ranseeds(1,2)
    kran0=ranseeds(1,3)
    mran0=ranseeds(1,4)
    nran0=ranseeds(1,5)
    rans=nran0
end if
if (length > 1) then
    Point to vector seeds.
    iran => ranseeds(2:,1)
    jran => ranseeds(2:,2)
    kran => ranseeds(2:,3)
    mran => ranseeds(2:,4)
    nran => ranseeds(2:,5)
    ranv = nran
```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
end if
lenran=length
END SUBROUTINE ran_init
```

f90 $hgt=hg \dots \text{if } (hgt+1 \neq hgng)$ Bit of dirty laundry here! We are testing whether the most positive integer hg wraps around to the most negative integer $hgng$ when 1 is added to it. We can't just write $hg+1$, since some compilers will evaluate this at compile time and return an overflow error message. If your compiler sees through the charade of the temporary variable hgt , you'll have to find another way to trick it.

$amm=\text{nearest}(1.0_sp,-1.0_sp)/hgng\dots$ Logically, amm should be a parameter; but the `nearest` intrinsic is trouble-prone in the initialization expression for a parameter (named constant), so we compute this at run time. We then check that amm , when multiplied by the largest possible negative integer, does not equal or exceed unity. (Our random deviates are guaranteed never to equal zero or unity exactly.)

You might wonder why amm is negative, and why we multiply it by negative integers to get positive random deviates. The answer, which will become manifest in the random generators given below, is that we want to use the fast `not` operation on integers to convert them to nonzero values of all one sign. This is possible if the conversion is to negative values, since `not(i)` is negative for all nonnegative i . If the conversion were to positive values, we would have problems both with zero (its sign bit is already positive) and $hgng$ (since `not(hgng)` is generally zero).

```
iran0=ranseeds(1,1) ...
iran => ranseeds(2:,1)...
```

The initial state information is stored in `ranseeds`, a two-dimensional array whose column (second) index ranges from 1 to 5 over the state variables. `ranseeds(1, :)` is reserved for scalar random generators, while `ranseeds(2: , :)` is for vector-parallel generators. The `ranseeds` array is made available to vector generators through the pointers `iran`, `jran`, `kran`, `mran`, and `nran`. The corresponding scalar values, `iran0, ..., nran0` are simply global variables, not pointers, because the overhead of addressing a scalar through a pointer is often too great. (We will have to copy these scalar values back into `ranseeds` when it, rarely, needs to be addressed as an array.)

`call ran_hash(...)` Unique, and random, initial state information is obtained by putting a user-settable “sequence number” into `iran`, a component number into `jran`, and hashing this pair. Then `jran` and `kran` are hashed, `kran` and `mran` are hashed, and so forth.

```
SUBROUTINE ran_deallocate
  User interface to release the workspace used by the random number routines.
  if (lenran > 0) then
    deallocate(ranseeds,ravn)
    nullify(ranseeds,ravn,iran,jran,kran,mran,nran)
    lenran = 0
  end if
END SUBROUTINE ran_deallocate
```

The above routine is supplied as a user interface for deallocating all the state space storage.

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
 Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
 Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

SUBROUTINE ran_seed(sequence,size,put,get)
IMPLICIT NONE
INTEGER, OPTIONAL, INTENT(IN) :: sequence
INTEGER, OPTIONAL, INTENT(OUT) :: size
INTEGER, DIMENSION(:), OPTIONAL, INTENT(IN) :: put
INTEGER, DIMENSION(:), OPTIONAL, INTENT(OUT) :: get
    User interface for seeding the random number routines. Syntax is exactly like Fortran 90's
    random_seed routine, with one additional argument keyword: sequence, set to any integer
    value, causes an immediate new initialization, seeded by that integer.
if (present(size)) then
    size=5*lenran
else if (present(put)) then
    if (lenran == 0) RETURN
    ranseeds=reshape(put,shape(ranseeds))
    where (ranseeds(:,1:3) < 0) ranseeds(:,1:3)=not(ranseeds(:,1:3))
        Enforce nonnegativity and nonzero conditions on any user-supplied seeds.
    where (ranseeds(:,4:5) == 0) ranseeds(:,4:5)=1
    iran0=ranseeds(1,1)
    jran0=ranseeds(1,2)
    kran0=ranseeds(1,3)
    mran0=ranseeds(1,4)
    nran0=ranseeds(1,5)
else if (present(get)) then
    if (lenran == 0) RETURN
    ranseeds(1,1:5)=(/ iran0,jran0,kran0,mran0,nran0 /)
    get=reshape(ranseeds,shape(get))
else if (present(sequence)) then
    call ran_deallocate
    seq=sequence
end if
END SUBROUTINE ran_seed

```

f90 ranseeds=reshape(put,shape(ranseeds)) ...
 get=reshape(ranseeds,shape(get))

Fortran 90's convention is that random state space is a one-dimensional array, so we map to this on both the get and put keywords.

```

iran0=...jran0=...kran0=...
ranseeds(1,1:5)=(/ iran0,jran0,kran0,mran0,nran0 /)

```

It's much more convenient to set a vector from a bunch of scalars then the other way around.

```

SUBROUTINE ran_hash_s(il,ir)
IMPLICIT NONE
INTEGER(K4B), INTENT(INOUT) :: il,ir
    DES-like hashing of two 32-bit integers, using shifts, xor's, and adds to make the internal
    nonlinear function.
INTEGER(K4B) :: is,j
do j=1,4
    is=ir
    ir=ieor(ir,ishft(ir,5))+1422217823
    ir=ieor(ir,ishft(ir,-16))+1842055030
    ir=ieor(ir,ishft(ir,9))+80567781
    ir=ieor(il,ir)
    il=is
end do
END SUBROUTINE ran_hash_s

```

The various constants are chosen to give
 good bit mixing and should not be
 changed.

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
 Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
 Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
 readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
<http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

SUBROUTINE ran_hash_v(il,ir)
IMPLICIT NONE
INTEGER(K4B), DIMENSION(:, ), INTENT(INOUT) :: il,ir
  Vector version of ran_hash_s.
INTEGER(K4B), DIMENSION(size(il)) :: is
INTEGER(K4B) :: j
do j=1,4
  is=ir
  ir=ieor(ir,ishft(ir,5))+1422217823
  ir=ieor(ir,ishft(ir,-16))+1842055030
  ir=ieor(ir,ishft(ir,9))+80567781
  ir=ieor(il,ir)
  il=is
end do
END SUBROUTINE ran_hash_v

```

END MODULE ran_state

The lines

```

ir=ieor(ir,ishft(ir,5))+1422217823
ir=ieor(ir,ishft(ir,-16))+1842055030
ir=ieor(ir,ishft(ir,9))+80567781

```

are *not* a Marsaglia shift sequence, though they resemble one. Instead, they implement a fast, nonlinear function on `ir` that we use as the “S-box” in a DES-like hashing algorithm. (See Volume 1, §7.5.) The triplet (5, -16, 9) is *not* chosen to give a full period Marsaglia sequence — it doesn’t. Instead it is chosen as being particularly good at separating in Hamming distance (i.e., number of nonidentical bits) two initially close values of `ir` (e.g., differing by only one bit). The large integer constants are chosen by a similar criterion. Note that the wrap-around of addition without generating an overflow error condition, which was tested in `ran_init`, is relied upon here.

★ ★ ★

```

SUBROUTINE ran0_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,iran0,jran0,kran0,nran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
  Lagged Fibonacci generator combined with a Marsaglia shift sequence. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90’s random_number routine. Use ran_seed to initialize or reinitialize to a particular sequence. The period of this generator is about  $2.0 \times 10^{28}$ , and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.
if (lenran < 1) call ran_init(1)                                Initialization routine in ran_state.
rans=iran0-kran0                                              Update Fibonacci generator, which
if (rans < 0) rans=rans+2147483579_k4b                         has period  $p^2 + p + 1$ ,  $p = 2^{31} - 69$ .
iran0=jran0
jran0=kran0
kran0=rans
nran0=ieor(nran0,ishft(nran0,13))                               Update Marsaglia shift sequence with
nran0=ieor(nran0,ishft(nran0,-17))                             period  $2^{32} - 1$ .
nran0=ieor(nran0,ishft(nran0,5))
rans=ieor(nran0,rans)
harvest=amm*merge(rans,not(rans), rans<0 )
END SUBROUTINE ran0_s

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

SUBROUTINE ran0_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B, amm, lenran, ran_init, iran, jran, kran, nran, ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
ranv(1:n)=ieor(nran(1:n),ranv(1:n))
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran0_v

```

This is the simplest, and fastest, of the generators provided. It combines a subtractive Fibonacci generator (Number 6 in ref. [1], and one of the generators in Marsaglia and Zaman's *mzran*) with a Marsaglia shift sequence. On typical machines it is only 20% or so faster than *ran1*, however; so we recommend the latter preferentially. While we know of no weakness in *ran0*, we are not offering a prize for finding a weakness. *ran0* does have the feature, useful if you have a machine with nonstandard arithmetic, that it does not go beyond Fortran 90's assumed integer model.

Note that *ran0_s* and *ran0_v* are overloaded by the module *nr* onto the single name *ran0* (and similarly for the routines below).

★ ★ ★

```

SUBROUTINE ran1_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B, amm, lenran, ran_init, &
    iran0, jran0, kran0, nran0, mran0, rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
Lagged Fibonacci generator combined with two Marsaglia shift sequences. On output, returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90's random_number routine. Use ran_seed to initialize or reinitialize to a particular sequence. The period of this generator is about  $8.5 \times 10^{37}$ , and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.
if (lenran < 1) call ran_init(1)                                Initialization routine in ran_state.
rans=iran0-kran0                                              Update Fibonacci generator, which
if (rans < 0) rans=rans+2147483579_k4b                         has period  $p^2 + p + 1$ ,  $p = 2^{31} - 69$ .
iran0=jran0
jran0=kran0
kran0=rans
nran0=ieor(nran0,ishft(nran0,13))                               Update Marsaglia shift sequence.
nran0=ieor(nran0,ishft(nran0,-17))
nran0=ieor(nran0,ishft(nran0,5))

Once only per cycle, advance sequence by 1, shortening its period to  $2^{32} - 2$ .
if (nran0 == 1) nran0=270369_k4b
mran0=ieor(mran0,ishft(mran0,5))                               Update Marsaglia shift sequence with
mran0=ieor(mran0,ishft(mran0,-13))                            period  $2^{32} - 1$ .
mran0=ieor(mran0,ishft(mran0,6))

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
rans=ieor(nran0,rans)+mran0
  Combine the generators. The above statement has wrap-around addition.
harvest=amm*merge(rans,not(rans), rans<0 )      Make the result positive definite (note
END SUBROUTINE ran1_s                           that amm is negative).
```

```
SUBROUTINE ran1_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
     iran,jran,kran,nran,mran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
where (nran(1:n) == 1) nran(1:n)=270369_k4b
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),5))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),-13))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),6))
ranv(1:n)=ieor(nran(1:n),ranv(1:n))+mran(1:n)
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran1_v
```

The routine `ran1` combines *three* fast generators: the two used in `ran0`, plus an additional (different) Marsaglia shift sequence. The last generator is combined via an addition that can wrap-around.

We think that, within the limits of its floating-point precision, `ran1` provides perfect random numbers. We will pay \$1000 to the first reader who convinces us otherwise (by exhibiting a statistical test that `ran1` fails in a nontrivial way, excluding the ordinary limitations of a floating-point representation).

* * *

```
SUBROUTINE ran2_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
     iran0,jran0,kran0,nran0,mran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
Lagged Fibonacci generator combined with a Marsaglia shift sequence and a linear congruential generator. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90's random_number routine. Use ran_seed to initialize or reinitialize to a particular sequence. The period of this generator is about  $8.5 \times 10^{37}$ , and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.
if (lenran < 1) call ran_init(1)           Initialization routine in ran_state.
rans=iran0-kran0                         Update Fibonacci generator, which
if (rans < 0) rans=rans+2147483579_k4b    has period  $p^2 + p + 1$ ,  $p = 2^{31} - 69$ .
iran0=jran0
jran0=kran0
kran0=rans
```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

nran0=ieor(nran0,ishft(nran0,13))          Update Marsaglia shift sequence with
nran0=ieor(nran0,ishft(nran0,-17))          period  $2^{32} - 1$ .
nran0=ieor(nran0,ishft(nran0,5))
rans=iand(mran0,65535)
      Update the sequence  $m \leftarrow 69069m + 820265819 \bmod 2^{32}$  using shifts instead of multiplies.
      Wrap-around addition (tested at initialization) is used.
mran0=ishft(3533*ishft(mran0,-16)+rans,16)+ &
      3533*rans+820265819_k4b
rans=ieor(nran0,kran0)+mran0
harvest=amm*merge(rans,not(rans), rans<0 )
END SUBROUTINE ran2_s
      Combine the generators.
      Make the result positive definite (note
      that amm is negative).

```

```

SUBROUTINE ran2_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
     iran,jran,kran,nran,mran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
ranv(1:n)=iand(mran(1:n),65535)
mran(1:n)=ishft(3533*ishft(mran(1:n),-16)+ranv(1:n),16)+ &
      3533*ranv(1:n)+820265819_k4b
ranv(1:n)=ieor(nran(1:n),kran(1:n))+mran(1:n)
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran2_v

```

ran2, for use by readers whose caution is extreme, also combines three generators. The difference from ran1 is that each generator is based on a completely different method from the other two. The third generator, in this case, is a linear congruential generator, modulo 2^{32} . This generator relies extensively on wrap-around addition (which is automatically tested at initialization). On machines with fast arithmetic, ran2 is on the order of only 20% slower than ran1. We offer a \$1000 bounty on ran2, with the same terms as for ran1, above.

★ ★ ★

```

SUBROUTINE expdev_s(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
      Returns in harvest an exponentially distributed, positive, random deviate of unit mean,
      using ran1 as the source of uniform deviates.
REAL(SP) :: dum
call ran1(dum)
harvest=-log(dum)           We use the fact that ran1 never returns exactly 0 or 1.
END SUBROUTINE expdev_s

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
SUBROUTINE expdev_v(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
REAL(SP), DIMENSION(size(harvest)) :: dum
call ran1(dum)
harvest=-log(dum)
END SUBROUTINE expdev_v
```

f90 call ran1(dum) The only noteworthy thing about this line is its simplicity: Once all the machinery is in place, the random number generators are self-initializing (to the sequence defined by seq = 0), and (via overloading) usable with both scalar and vector arguments.

★ ★ ★

```
SUBROUTINE gasdev_s(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
    Returns in harvest a normally distributed deviate with zero mean and unit variance, using
    ran1 as the source of uniform deviates.
REAL(SP) :: rsq,v1,v2
REAL(SP), SAVE :: g
LOGICAL, SAVE :: gaus_stored=.false.
if (gaus_stored) then
    harvest=g
    gaus_stored=.false.
else
    do
        call ran1(v1)
        call ran1(v2)
        v1=2.0_sp*v1-1.0_sp
        v2=2.0_sp*v2-1.0_sp
        rsq=v1**2+v2**2
        if (rsq > 0.0 .and. rsq < 1.0) exit
    end do
    rsq=sqrt(-2.0_sp*log(rsq)/rsq)
    harvest=v1*rsq
    g=v2*rsq
    gaus_stored=.true.
end if
END SUBROUTINE gasdev_s
```

We have an extra deviate handy,
so return it,
and unset the flag.
We don't have an extra deviate handy, so
pick two uniform numbers in the square ex-
tending from -1 to +1 in each direction,
see if they are in the unit circle,
otherwise try again.
Now make the Box-Muller transformation to
get two normal deviates. Return one and
save the other for next time.
Set flag.

```
SUBROUTINE gasdev_v(harvest)
USE nrtype; USE nrutil, ONLY : array_copy
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
REAL(SP), DIMENSION(size(harvest)) :: rsq,v1,v2
REAL(SP), ALLOCATABLE, DIMENSION(:), SAVE :: g
INTEGER(I4B) :: n,ng,nn,m
INTEGER(I4B), SAVE :: last_allocated=0
LOGICAL, SAVE :: gaus_stored=.false.
LOGICAL, DIMENSION(size(harvest)) :: mask
n=size(harvest)
if (n /= last_allocated) then
```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
<http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

if (last_allocated /= 0) deallocate(g)
allocate(g(n))
last_allocated=n
gaus_stored=.false.
end if
if (gaus_stored) then
  harvest=g
  gaus_stored=.false.
else
  ng=1
  do
    if (ng > n) exit
    call ran1(v1(ng:n))
    call ran1(v2(ng:n))
    v1(ng:n)=2.0_sp*v1(ng:n)-1.0_sp
    v2(ng:n)=2.0_sp*v2(ng:n)-1.0_sp
    rsq(ng:n)=v1(ng:n)**2+v2(ng:n)**2
    mask(ng:n)=(rsq(ng:n)>0.0 .and. rsq(ng:n)<1.0)
    call array_copy(pack(v1(ng:n),mask(ng:n)),v1(ng:),nn,m)
    v2(ng:ng+nn-1)=pack(v2(ng:n),mask(ng:n))
    rsq(ng:ng+nn-1)=pack(rsq(ng:n),mask(ng:n))
    ng=ng+nn
  end do
  rsq=sqrt(-2.0_sp*log(rsq)/rsq)
  harvest=v1*rsq
  g=v2*rsq
  gaus_stored=.true.
end if
END SUBROUTINE gasdev_v

```

 if (n /= last_allocated) ... We make the assumption that, in most cases, the size of harvest will not change between successive calls. Therefore, if it *does* change, we don't try to save the previously generated deviates that, half the time, will be around. If your use has rapidly varying sizes (or, even worse, calls alternating between two different sizes), you should remedy this inefficiency in the obvious way.

call array_copy(pack(v1(ng:n),mask(ng:n)),v1(ng:),nn,m) This is a variant of the pack-unpack method (see note to `factrl`, p. 1087). Different here is that we don't care which random deviates end up in which component. Thus, we can simply keep packing successful returns into `v1` and `v2` until they are full.

 Note also the use of `array_copy`, since we don't know in advance the length of the array returned by `pack`.

* * *

```

FUNCTION gamdev(ia)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ia
REAL(SP) :: gamdev
Returns a deviate distributed as a gamma distribution of integer order ia, i.e., a waiting
time to the ia-th event in a Poisson process of unit mean, using ran1 as the source of
uniform deviates.
REAL(SP) :: am,e,h,s,x,y,v(2),arr(5)
call assert(ia >= 1, 'gamdev arg')
if (ia < 6) then
  Use direct method, adding waiting times.

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
<http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

call ran1(arr(1:ia))
x=-log(product(arr(1:ia)))
else                                Use rejection method.
do
  call ran1(v)
  v(2)=2.0_sp*v(2)-1.0_sp           These three lines generate the tangent of a
  if (dot_product(v,v) > 1.0) cycle   random angle, i.e., are equivalent to
  y=v(2)/v(1)                         y = tan( $\pi$ ran(idum)).
  am=ia-1
  s=sqrt(2.0_sp*am+1.0_sp)
  x=s*y+am                           We decide whether to reject x:
  if (x <= 0.0) cycle               Reject in region of zero probability.
  e=(1.0_sp+y**2)*exp(am*log(x/am)-s*y)      Ratio of probability function to
  call ran1(h)                         comparison function.
  if (h <= e) exit                  Reject on basis of a second uniform deviate.
end do
end if
gamdev=x
END FUNCTION gamdev

```



`x=-log(product(arr(1:ia)))` Why take the log of the product instead of the sum of the logs? Because log is assumed to be slower than multiply.



We don't have vector versions of the less commonly used deviate generators, `gamdev`, `poidev`, and `bnldev`.

* * *

```

FUNCTION poidev(xm)
USE nrtype
USE nr, ONLY : gammln,ran1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xm
REAL(SP) :: poidev
  Returns as a floating-point number an integer value that is a random deviate drawn from a
  Poisson distribution of mean xm, using ran1 as a source of uniform random deviates.
REAL(SP) :: em,harvest,t,y
REAL(SP), SAVE :: alxm,g,oldm=-1.0_sp,sq
  oldm is a flag for whether xm has changed since last call.
if (xm < 12.0) then                Use direct method.
  if (xm /= oldm) then
    oldm=xm
    g=exp(-xm)                      If xm is new, compute the exponential.
  end if
  em=-1
  t=1.0
  do
    em=em+1.0_sp
    call ran1(harvest)
    t=t*harvest
    if (t <= g) exit
  end do
else
  if (xm /= oldm) then
    oldm=xm
    sq=sqrt(2.0_sp*xm)
    alxm=log(xm)
    g=xm*alxm-gammln(xm+1.0_sp)
  end if
  do

```

Instead of adding exponential deviates it is equivalent to multiply uniform deviates.
We never actually have to take the log; merely compare to the pre-computed exponential.

Use rejection method.
If xm has changed since the last call, then precompute some functions that occur below.

The function `gammln` is the natural log of the gamma function, as given in §6.1.

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

do
    call ran1(harvest)
    y=tan(PI*harvest)
    em=sq*y+xm
    if (em >= 0.0) exit
end do
em=int(em)
t=0.9_sp*(1.0_sp+y**2)*exp(em*alxm-gammln(em+1.0_sp)-g)
The ratio of the desired distribution to the comparison function; we accept or reject
by comparing it to another uniform deviate. The factor 0.9 is chosen so that t never
exceeds 1.
call ran1(harvest)
if (harvest <= t) exit
end do
end if
poidev=em
END FUNCTION poidev

```

y is a deviate from a Lorentzian comparison
function.
em is y, shifted and scaled.
Reject if in regime of zero probability.

The trick for integer-valued distributions.

★ ★ ★

```

FUNCTION bnldev(pp,n)
USE nrtype
USE nr, ONLY : gammln,ran1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: pp
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: bnldev
    Returns as a floating-point number an integer value that is a random deviate drawn from a binomial distribution of n trials each of probability pp, using ran1 as a source of uniform random deviates.
INTEGER(I4B) :: j
INTEGER(I4B), SAVE :: nold=-1
REAL(SP) :: am,em,g,h,p,sq,t,y,arr(24)
REAL(SP), SAVE :: pc,plog,pclog,en,oldg,pold=-1.0      Arguments from previous calls.
p=merge(pp,1.0_sp-pp, pp <= 0.5_sp )
    The binomial distribution is invariant under changing pp to 1.-pp, if we also change the answer to n minus itself; we'll remember to do this below.
am=n*p
if (n < 25) then
    call ran1(arr(1:n))
    bnldev=count(arr(1:n)<p)
else if (am < 1.0) then
    g=exp(-am)
    t=1.0
    do j=0,n
        call ran1(h)
        t=t*h
        if (t < g) exit
    end do
    bnldev=merge(j,n, j <= n)
else
    if (n /= nold) then
        en=n
        oldg=gammln(en+1.0_sp)
        nold=n
    end if
    if (p /= pold) then
        pc=1.0_sp-p
        plog=log(p)
        pclog=log(pc)
        pold=p
    end if
    if (n > 25) then
        This is the mean of the deviate to be produced.
        Use the direct method while n is not too large.
        This can require up to 25 calls to ran1.
        If fewer than one event is expected out of 25 or more trials, then the distribution is quite accurately Poisson. Use direct Poisson method.
        Use the rejection method.
        If n has changed, then compute useful quantities.
        If p has changed, then compute useful quantities.
    end if
end function bnldev

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0). Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

end if
sq=sqrt(2.0_sp*am*pc)           The following code should by now seem familiar:
do                                rejection method with a Lorentzian compar-
    call ran1(h)                  ison function.

    y=tan(PI*h)
    em=sq*y+am
    if (em < 0.0 .or. em >= em+1.0_sp) cycle      Reject.
    em=int(em)                   Trick for integer-valued distribution.
    t=1.2_sp*sq*(1.0_sp+y**2)*exp(oldg-gammln(em+1.0_sp)-&
        gammln(em-em+1.0_sp)+em*plog+(en-em)*pclog)
    call ran1(h)
    if (h <= t) exit            Reject. This happens about 1.5 times per devi-
    ate, on average.
end do
bnldev=em
end if
if (p /= pp) bnldev=n-bnldev      Remember to undo the symmetry transforma-
END FUNCTION bnldev              tion.

```

★ ★ ★

f90 The routines `psdes` and `psdes_safe` both perform *exactly* the same hashing as was done by the Fortran 77 routine `psdes`. The difference is that `psdes` makes assumptions about arithmetic that go beyond the strict Fortran 90 model, while `psdes_safe` makes no such assumptions. The disadvantage of `psdes_safe` is that it is significantly slower, performing most of its arithmetic in double-precision reals that are then converted to integers with Fortran 90's `modulo` intrinsic.

In fact the nonsafe version, `psdes`, works fine on almost all machines and compilers that we have tried. There is a reason for this: Our assumed integer model is the same as the C language `unsigned int`, and virtually all modern computers and compilers have a lot of C hidden inside. If `psdes` and `psdes_safe` produce identical output on your system for any hundred or so different input values, you can be quite confident about using the faster version exclusively.

At the other end of things, note that in the very unlikely case that your system fails on the `ran_hash` routine in the `ran_state` module (you will have learned this from error messages generated by `ran_init`), you can substitute `psdes_safe` for `ran_hash`: They are plug-compatible.

```

SUBROUTINE psdes_s(lword,rword)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
  "Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are
  returned hashed on all bits. Note that this version of the routine assumes properties of
  integer arithmetic that go beyond the Fortran 90 model, though they are compatible with
  unsigned integers in C.
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B) :: i,ia,ib,iswap,itmph,itmpl
do i=1,NITER          Perform niter iterations of DES logic, using a simpler
    iswap=rword          (noncryptographic) nonlinear function instead of DES's.
    ia=ieor(rword,C1(i)) The bit-rich constants C1 and (below) C2 guarantee lots
    itmpl=iand(ia,65535) of nonlinear mixing.
    itmph=iand(ishft(ia,-16),65535)

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

ib=itmpl**2+not(itmph**2)
ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
rword=ieor(lword,ieor(C2(i),ia)+itmpl*itmph)
lword=iswap
end do
END SUBROUTINE psdes_s

SUBROUTINE psdes_v(lword,rword)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:, INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B), DIMENSION(size(lword)) :: ia,ib,iswap,itmph,itmpl
INTEGER(I4B) :: i
i(assert_eq(size(lword),size(rword),'psdes_v')
do i=1,NITER
    iswap=rword
    ia=ieor(rword,C1(i))
    itmpl=iand(ia,65535)
    itmph=iand(ishft(ia,-16),65535)
    ib=itmpl**2+not(itmph**2)
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,ieor(C2(i),ia)+itmpl*itmph)
    lword=iswap
end do
END SUBROUTINE psdes_v

```

```

SUBROUTINE psdes_safe_s(lword,rword)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
    "Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are
    returned hashed on all bits. This is a slower version of the routine that makes no assumptions
    outside of the Fortran 90 integer model.
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B) :: i,ia,ib,iswap
REAL(DP) :: alo,ahi
do i=1,NITER
    iswap=rword
    ia=ieor(rword,C1(i))
    alo=real(iand(ia,65535),dp)
    ahi=real(iand(ishft(ia,-16),65535),dp)
    ib=modint(alo*alo+real(not(modint(ahi*ahi)),dp))
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,modint(real(ieor(C2(i),ia),dp)+alo*ahi))
    lword=iswap
end do
CONTAINS
FUNCTION modint(x)
REAL(DP), INTENT(IN) :: x
INTEGER(I4B) :: modint
REAL(DP) :: a
REAL(DP), PARAMETER :: big=huge(modint), base=big+big+2.0_dp
a=modulo(x,base)

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

if (a > big) a=a-base
modint=nint(a,kind=i4b)
END FUNCTION modint
END SUBROUTINE psdes_safe_s

SUBROUTINE psdes_safe_v(lword,rword)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
INTEGER(I4B), SAVE :: C1(4),C2(4)
DATA C1 /Z'BA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B), DIMENSION(size(lword)) :: ia,ib,iswap
REAL(DP), DIMENSION(size(lword)) :: alo,ahi
INTEGER(I4B) :: i
i(assert_eq(size(lword),size(rword),'psdes_safe_v'))
do i=1,NITER
    iswap=rword
    ia=ieor(rword,C1(i))
    alo=real(iand(ia,65535),dp)
    ahi=real(iand(ishft(ia,-16),65535),dp)
    ib=modint(alo*alo+real(not(modint(ahi*ahi)),dp))
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,modint(real(ieor(C2(i),ia),dp)+alo*ahi))
    lword=iswap
end do
CONTAINS

FUNCTION modint(x)
REAL(DP), DIMENSION(:), INTENT(IN) :: x
INTEGER(I4B), DIMENSION(size(x)) :: modint
REAL(DP), DIMENSION(size(x)) :: a
REAL(DP), PARAMETER :: big=huge(modint), base=big+big+2.0_dp
a=modulo(x,base)
where (a > big) a=a-base
modint=nint(a,kind=i4b)
END FUNCTION modint
END SUBROUTINE psdes_safe_v

```



FUNCTION modint(x) This embedded routine takes a double-precision real argument, and returns it as an integer mod 2^{32} (correctly wrapping it to negative to take into account that Fortran 90 has no unsigned integers).

* * *

```

SUBROUTINE ran3_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,ran_hash,mran0,nran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
Random number generation by DES-like hashing of two 32-bit words, using the algorithm
ran_hash. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive
of the endpoint values).
INTEGER(K4B) :: temp
if (lenran < 1) call ran_init(1)
nran0=ieor(nran0,ishft(nran0,13))
nran0=ieor(nran0,ishft(nran0,-17))
nran0=ieor(nran0,ishft(nran0,5))
if (nran0 == 1) nran0=270369_k4b

```

Initialize.
Two Marsaglia shift sequences are maintained as input to the hashing. The period of the combined generator is about 1.8×10^{19} .

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
<http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

rans=nran0
mran0=ieor(mran0,ishft(mran0,5))
mran0=ieor(mran0,ishft(mran0,-13))
mran0=ieor(mran0,ishft(mran0,6))
temp=mran0
call ran_hash(temp,rans)           Hash.
harvest=amm*merge(rans,not(rans), rans<0 )   Make the result positive definite (note
                                              that amm is negative).
END SUBROUTINE ran3_s

```

```

SUBROUTINE ran3_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,ran_hash,mran,nran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B), DIMENSION(size(harvest)) :: temp
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
where (nran(1:n) == 1) nran(1:n)=270369_k4b
ranv(1:n)=nran(1:n)
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),5))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),-13))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),6))
temp=mran(1:n)
call ran_hash(temp,ranv(1:n))
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran3_v

```

As given, `ran3` uses the `ran_hash` function in the module `ran_state` as its DES surrogate. That function is sufficiently fast to make `ran3` only about a factor of 2 slower than our baseline recommended generator `ran1`. The slower routine `psdes` and (even slower) `psdes_safe` are plug-compatible with `ran_hash`, and could be substituted for it in this routine.

★ ★ ★

```

FUNCTION irbit1(iseed)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit1
  Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
  is modified for the next call).
if (btest(iseed,17) .neqv. btest(iseed,4) .neqv. btest(iseed,1) &
  .neqv. btest(iseed,0)) then
  iseed=ibset(ishft(iseed,1),0)      Leftshift the seed and put a 1 in its bit 1.
  irbit1=1
else
  iseed=ishft(iseed,1)               But if the XOR calculation gave a 0,
  irbit1=0                           then put that in bit 1 instead.
end if
END FUNCTION irbit1

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-
readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
<http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

FUNCTION irbit2(iseed)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit2
    Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
    is modified for the next call).
INTEGER(I4B), PARAMETER :: IB1=1,IB2=2,IB5=16,MASK=IB1+IB2+IB5
if (btest(iseed,17)) then           Change all masked bits, shift, and put 1 into bit 1.
    iseed=ibset(ishft(ieor(iseed,MASK),1),0)
    irbit2=1
else                                Shift and put 0 into bit 1.
    iseed=ibclr(ishft(iseed,1),0)
    irbit2=0
end if
END FUNCTION irbit2

```

★ ★ ★

```

SUBROUTINE sobseq(x,init)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), OPTIONAL, INTENT(IN) :: init
INTEGER(I4B), PARAMETER :: MAXBIT=30,MAXDIM=6
    When the optional integer init is present, internally initializes a set of MAXBIT direction
    numbers for each of MAXDIM different Sobol' sequences. Otherwise returns as the vector x
    of length N the next values from N of these sequences. (N must not be changed between
    initializations.)
REAL(SP), SAVE :: fac
INTEGER(I4B) :: i,im,ipp,j,k,l
INTEGER(I4B), DIMENSION(:, :, ALLOCATABLE::: iu
INTEGER(I4B), SAVE :: in
INTEGER(I4B), DIMENSION(MAXDIM), SAVE :: ip,ix,mdeg
INTEGER(I4B), DIMENSION(MAXDIM*MAXBIT), SAVE :: iv
DATA ip /0,1,1,2,1,4/, mdeg /1,2,3,3,4,4/, ix /6*0/
DATA iv /6*1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9,156*0/
if (present(init)) then           Initialize, don't return a vector.
    ix=0
    in=0
    if (iv(1) /= 1) RETURN
    fac=1.0_sp/2.0_sp**MAXBIT
    allocate(iu(MAXDIM,MAXBIT))
    iu=reshape(iv,shape(iu))          To allow both 1D and 2D addressing.
    do k=1,MAXDIM
        do j=1,mdeg(k)             Stored values require only normalization.
            iu(k,j)=iu(k,j)*2**((MAXBIT-j)
        end do
        do j=mdeg(k)+1,MAXBIT       Use the recurrence to get other values.
            ipp=ip(k)
            i=iu(k,j-mdeg(k))
            i=ieor(i,i/2**mdeg(k))
            do l=mdeg(k)-1,1,-1
                if (btest(ipp,0)) i=ieor(i,iu(k,j-l))
                ipp=ipp/2
            end do
            iu(k,j)=i
        end do
    end do
    iv=reshape(iu,shape(iv))
    deallocate(iu)

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

else                                Calculate the next vector in the sequence.
  im=in
  do j=1,MAXBIT                   Find the rightmost zero bit.
    if (.not. btest(im,0)) exit
    im=im/2
  end do
  if (j > MAXBIT) call nrerror('MAXBIT too small in sobseq')
  im=(j-1)*MAXDIM
  j=min(size(x),MAXDIM)
  ix(1:j)=ieor(ix(1:j),iv(1+im:j+im))
    XOR the appropriate direction number into each component of the vector and convert
    to a floating number.
  x(1:j)=ix(1:j)*fac
  in=in+1                           Increment the counter.
end if
END SUBROUTINE sobseq

```

f90 if (present(init)) then ... allocate(iu(...)) ... iu=reshape(...)
 Wanting to avoid the deprecated EQUIVALENCE statement, we must
 reshape iv into a two-dimensional array, then un-reshape it after we
 are done. This is done only once, at initialization time, so there is no serious
 inefficiency introduced.

* * *

```

SUBROUTINE vegas(region,func,init,ncall,itmx,nprn,tgral,sd,chi2a)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: region
INTEGER(I4B), INTENT(IN) :: init,ncall,itmx,nprn
REAL(SP), INTENT(OUT) :: tgral,sd,chi2a
INTERFACE
  FUNCTION func(pt,wgt)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), DIMENSION(:), INTENT(IN) :: pt
  REAL(SP), INTENT(IN) :: wgt
  REAL(SP) :: func
  END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: ALPH=1.5_sp,TINY=1.0e-30_sp
INTEGER(I4B), PARAMETER :: MXDIM=10,NDMX=50

```

Performs Monte Carlo integration of a user-supplied d -dimensional function func over a rectangular volume specified by region, a vector of length $2d$ consisting of d "lower left" coordinates of the region followed by d "upper right" coordinates. The integration consists of itmx iterations, each with approximately ncall calls to the function. After each iteration the grid is refined; more than 5 or 10 iterations are rarely useful. The input flag init signals whether this call is a new start, or a subsequent call for additional iterations (see comments below). The input flag nprn (normally 0) controls the amount of diagnostic output. Returned answers are tgral (the best estimate of the integral), sd (its standard deviation), and chi2a (χ^2 per degree of freedom, an indicator of whether consistent results are being obtained). See text for further details.

```

INTEGER(I4B), SAVE :: i,it,j,k,mds,nd,ndim,ndo,ng,npg      Best make everything static,
INTEGER(I4B), DIMENSION(MXDIM), SAVE :: ia/kg             allowing restarts.
REAL(SP), SAVE :: calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,xo,harvest
REAL(SP), DIMENSION(NDMX,MXDIM), SAVE :: d,di,xi
REAL(SP), DIMENSION(MXDIM), SAVE :: dt,dx,x
REAL(SP), DIMENSION(NDMX), SAVE :: r,xin
REAL(DP), SAVE :: schi,si,swgt

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
 Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
 Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

ndim=size(region)/2
if (init <= 0) then
    mds=1
    ndo=1
    xi(1,:)=1.0
end if
if (init <= 1) then
    si=0.0
    swgt=0.0
    schi=0.0
end if
if (init <= 2) then
    nd=NDMX
    ng=1
    if (mds /= 0) then
        ng=(ncall/2.0_sp+0.25_sp)**(1.0_sp/ndim)
        mds=1
        if ((2*ng-NDMX) >= 0) then
            mds=-1
            npg=ng/NDMX+1
            nd=ng/npg
            ng=npg*nd
        end if
    end if
    k=ng**ndim
    npg=max(ncall/k,2)
    calls=real(npg,sp)*real(k,sp)
    dxg=1.0_sp/ng
    dv2g=(calls*dxg**ndim)**2/npg/npg/(npg-1.0_sp)
    xnd=nd
    dxg=dxg*xnd
    dx(1:ndim)=region(1+ndim:2*ndim)-region(1:ndim)
    xjac=1.0_sp/calls*product(dx(1:ndim))
    if (nd /= ndo) then
        r(1:max(nd,ndo))=1.0
        do j=1,ndim
            call rebin(ndo/xnd,nd,r,xin,xi(:,j))
        end do
        ndo=nd
    end if
    if (nprn >= 0) write(*,200) ndim,calls,it,itmx,nprn,&
        ALPH,mds,nd,(j,region(j),j,region(j+ndim),j=1,ndim)
end if
do it=1,itmx
    ti=0.0
    tsi=0.0
    kg(:)=1
    d(1:nd,:)={0.0}
    di(1:nd,:)=0.0
    iterate: do
        fb=0.0
        f2b=0.0
        do k=1,npg
            wgt=xjac
            do j=1,ndim
                call ran1(harvest)
                xn=(kg(j)-harvest)*dxg+1.0_sp
                ia(j)=max(min(int(xn),NDMX),1)
                if (ia(j) > 1) then
                    xo=xi(ia(j),j)-xi(ia(j)-1,j)
                    rc=xi(ia(j)-1,j)+(xn-ia(j))*xo
                else
                    xo=xi(ia(j),j)
                    rc=(xn-ia(j))*xo
                end if
                xi(j)=xo+rc
            end do
        end do
    end do
    if (it >= 3) then
        do j=1,ndim
            call rebin(ndo/xnd,nd,r,xin,xi(:,j))
        end do
    end if
    if (it >= 3) then
        do j=1,ndim
            call rebin(ndo/xnd,nd,r,xin,xi(:,j))
        end do
    end if
end do

```

Normal entry. Enter here on a cold start.
Change to `mds=0` to disable stratified sampling, i.e., use importance sampling only.

Enter here to inherit the grid from a previous call, but not its answers.

Enter here to inherit the previous grid and its answers.

Set up for stratification.

Main iteration loop. Can enter here (`init ≥ 3`) to do an additional `itm` iterations with all other parameters unchanged.

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0). Copyright (C) 1986–1996 by Cambridge University Press. Programs Copyright (C) 1986–1996 by Numerical Recipes Software. Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit www.nr.com or call 1-800-872-7423 (North America only), or send email to directcstusrv@cambridge.org (outside North America).

```

        end if
        x(j)=region(j)+rc*dx(j)
        wgt=wgt*xo*xnd
    end do
    f=wgt*func(x(1:ndim),wgt)
    f2=f*f
    fb=fb+f
    f2b=f2b+f2
    do j=1,ndim
        di(ia(j),j)=di(ia(j),j)+f
        if (mds >= 0) d(ia(j),j)=d(ia(j),j)+f2
    end do
    end do
    f2b=sqrt(f2b*npg)
    f2b=(f2b-fb)*(f2b+fb)
    if (f2b <= 0.0) f2b=TINY
    ti=ti+fb
    tsi=tsi+f2b
    if (mds < 0) then           Use stratified sampling.
        do j=1,ndim
            d(ia(j),j)=d(ia(j),j)+f2b
        end do
    end if
    do k=ndim,1,-1
        kg(k)=mod(kg(k),ng)+1
        if (kg(k) /= 1) cycle iterate
    end do
    exit iterate
end do iterate
tsi=tsi*dv2g                      Compute final results for this iteration.
wgt=1.0_sp/tsi
si=si+real(wgt,dp)*real(ti,dp)
schi=schi+real(wgt,dp)*real(ti,dp)**2
swgt=swgt+real(wgt,dp)
tgral=si/swgt
chi2a=max((schi-si*tgral)/(it-0.99_dp),0.0_dp)
sd=sqrt(1.0_sp/swgt)
tsi=sqrt(tsi)
if (nprn >= 0) then
    write(*,201) it,ti,tsi,tgral,sd,chi2a
    if (nprn /= 0) then
        do j=1,ndim
            write(*,202) j,(xi(i,j),di(i,j),&
                i=1+nprn/2,nd,nprn)
        end do
    end if
end if
do j=1,ndim                         Refine the grid. Consult references to under-
    xo=d(1,j)                        stand the subtlety of this procedure. The
    xn=d(2,j)                        refinement is damped, to avoid rapid,
    d(1,j)=(xo+xn)/2.0_sp             destabilizing changes, and also compressed
    dt(j)=d(1,j)                     in range by the exponent ALPH.
    do i=2,nd-1
        rc=xo+xn
        xo=xn
        xn=d(i+1,j)
        d(i,j)=(rc+xn)/3.0_sp
        dt(j)=dt(j)+d(i,j)
    end do
    d(nd,j)=(xo+xn)/2.0_sp
    dt(j)=dt(j)+d(nd,j)
end do
where (d(1:nd,:)< TINY) d(1:nd,:)=TINY
do j=1,ndim

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

r(1:nd)=((1.0_sp-d(1:nd,j)/dt(j))/(log(dt(j))-log(d(1:nd,j))))**ALPH
rc=sum(r(1:nd))
call rebin(rc/xnd,nd,r,xin,xi(:,j))
end do
end do
200 format(/' input parameters for vegas: ndim=',i3,' ncalls=',f8.0&
/28x,' it=',i5,' itmx=',i5&
/28x,' nprn=',i3,' alph=',f5.2/28x,' mds=',i3,' nd=',i4&
/(30x,'xl('i2,')= ',g11.4,' xu('i2,')= ',g11.4))
201 format(/' iteration no.',I3,': ','integral =',g14.7,' +/- ',g9.2,&
'/ all iterations: integral =',g14.7,' +/- ',g9.2,&
' chi**2/it''n = ',g9.2)
202 format(/' data for axis ',I2/' X delta i ',&
' x delta i ',' x delta i ',&
/(1x,f7.5,1x,g11.4,5x,f7.5,1x,g11.4,5x,f7.5,1x,g11.4))
CONTAINS
SUBROUTINE rebin(rc,nd,r,xin,xi)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: rc
INTEGER(I4B), INTENT(IN) :: nd
REAL(SP), DIMENSION(:), INTENT(IN) :: r
REAL(SP), DIMENSION(:), INTENT(OUT) :: xin
REAL(SP), DIMENSION(:), INTENT(INOUT) :: xi
      Utility routine used by vegas, to rebin a vector of densities xi into new bins defined by
      a vector r.
INTEGER(I4B) :: i,k
REAL(SP) :: dr,xn,xo
k=0
xo=0.0
dr=0.0
do i=1,nd-1
  do
    if (rc <= dr) exit
    k=k+1
    dr=dr+r(k)
  end do
  if (k > 1) xo=xi(k-1)
  xn=xi(k)
  dr=dr-rc
  xin(i)=xn-(xn-xo)*dr/r(k)
end do
xi(1:nd-1)=xin(1:nd-1)
xi(nd)=1.0
END SUBROUTINE rebin
END SUBROUTINE vegas

```

★ ★ ★

```

RECURSIVE SUBROUTINE miser(func,regn,ndim,npts,dith,ave,var)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTERFACE
  FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP) :: func
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
  END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), INTENT(IN) :: regn
INTEGER(I4B), INTENT(IN) :: ndim,npts

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

REAL(SP), INTENT(IN) :: dith
REAL(SP), INTENT(OUT) :: ave,var
REAL(SP), PARAMETER :: PFAC=0.1_sp,TINY=1.0e-30_sp,BIG=1.0e30_sp
INTEGER(I4B), PARAMETER :: MNPT=15,MNBS=60
Monte Carlo samples a user-supplied ndim-dimensional function func in a rectangular
volume specified by region, a  $2 \times \text{ndim}$  vector consisting of ndim "lower-left" coordinates
of the region followed by ndim "upper-right" coordinates. The function is sampled a total
of npts times, at locations determined by the method of recursive stratified sampling. The
mean value of the function in the region is returned as ave; an estimate of the statistical
uncertainty of ave (square of standard deviation) is returned as var. The input parameter
dith should normally be set to zero, but can be set to (e.g.) 0.1 if func's active region
falls on the boundary of a power-of-2 subdivision of region.
Parameters: PFAC is the fraction of remaining function evaluations used at each stage to
explore the variance of func. At least MNPT function evaluations are performed in any
terminal subregion; a subregion is further bisected only if at least MNBS function evaluations
are available.
REAL(SP), DIMENSION(:), ALLOCATABLE :: regn_temp
INTEGER(I4B) :: j,jb,n,ndum,npri,nppt,npqr
INTEGER(I4B), SAVE :: iran=0
REAL(SP) :: avel,varl,fracl,fval,rgl,rgm,rgr,&
           s,sigl,siglb,sigr,sigrb,sm,sm2,sumb,sumr
REAL(SP), DIMENSION(:), ALLOCATABLE :: fmaxl,fmaxr,fminl,fminr,pt,rmid
ndum=assert_eq(size(regn),2*ndim,'miser')
allocate(pt(ndim))
if (npts < MNBS) then
    sm=0.0
    sm2=0.0
    do n=1,npts
        call ranpt(pt,regn)
        fval=func(pt)
        sm=sm+fval
        sm2=sm2+fval**2
    end do
    ave=sm/npts
    var=max(TINY,(sm2-sm**2/npts)/npts**2)
else
    npri=max(int(npts*PFAC),MNPT)
    allocate(rmid(ndim),fmaxl(ndim),fmaxr(ndim),fminl(ndim),fminr(ndim))
    fminl(:)=BIG
    fminr(:)=BIG
    fmaxl(:)=-BIG
    fmaxr(:)=-BIG
    do j=1,ndim
        iran=mod(iran*2661+36979,175000)
        s=sign(dith,real(iran-87500,sp))
        rmid(j)=(0.5_sp+s)*regn(j)+(0.5_sp-s)*regn(ndim+j)
    end do
    do n=1,npri
        call ranpt(pt,regn)
        fval=func(pt)
        where (pt <= rmid)
            fminl=min(fminl,fval)
            fmaxl=max(fmaxl,fval)
        elsewhere
            fminr=min(fminr,fval)
            fmaxr=max(fmaxr,fval)
        end where
    end do
    sumb=BIG
    jb=0
    siglb=1.0
    sigrb=1.0
    do j=1,ndim
        if (fmaxl(j) > fminl(j) .and. fmaxr(j) > fminr(j)) then
            Too few points to bisect; do straight Monte
            Carlo.
            Do the preliminary (uniform) sampling.
            Initialize the left and right bounds for each
            dimension.
        end if
        if (siglb>0.0 .and. sigrb>0.0) then
            Loop over the points in the sample.
            Find the left and right bounds for each di-
            mension.
        end if
    end do
    Choose which dimension jb to bisect.
end if

```

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

sigl=max(TINY,(fmaxl(j)-fminl(j))**(2.0_sp/3.0_sp))
sigr=max(TINY,(fmaxr(j)-fminr(j))**(2.0_sp/3.0_sp))
sumr=sigl+sigr
if (sumr <= sumb) then
    sumb=sumr
    jb=j
    siglb=sigl
    sigrb=sigr
end if
end if
end do
deallocate(fminr,fminl,fmaxr,fmaxl)
if (jb == 0) jb=1+(ndim*iran)/175000      MNPT may be too small.
rgl=regn(jb)                                Apportion the remaining points between left
rgm=rmid(jb)                                and right.
rgr=regn(ndim+jb)
fracl=abs((rgm-rgl)/(rgr-rgl))
nptl=(MNPT+(npts-npre-2*MNPT)*fracl*siglb/ &   Equation (7.8.23).
      (fracl*siglb+(1.0_sp-fracl)*sigrb))
nptr=npts-npre-nptl
allocate(regn_temp(2*ndim))
regn_temp(:)=regn(:)
regn_temp(ndim+jb)=rmid(jb)                  Set region to left.
call miser(func,regn_temp,ndim,nptl,dith,ave1,var1)
      Dispatch recursive call; will return back here eventually.
regn_temp(jb)=rmid(jb)
regn_temp(ndim+jb)=regn(ndim+jb)      Set region to right.
call miser(func,regn_temp,ndim,nptr,dith,ave,var)
      Dispatch recursive call; will return back here eventually.
deallocate(regn_temp)
ave=fracl*ave1+(1-fracl)*ave      Combine left and right regions by equation
var=fracl*fracl*var1+(1-fracl)*var      (7.8.11) (1st line).
deallocate(rmid)
end if
deallocate(pt)
CONTAINS
SUBROUTINE ranpt(pt,region)
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: pt
REAL(SP), DIMENSION(:), INTENT(IN) :: region
Returns a uniformly random point pt in a rectangular region of dimension d. Used by
miser; calls ran1 for uniform deviates.
INTEGER(I4B) :: n
call ran1(pt)
n=size(pt)
pt(1:n)=region(1:n)+(region(n+1:2*n)-region(1:n))*pt(1:n)
END SUBROUTINE ranpt
END SUBROUTINE miser

```

f90 The Fortran 90 version of this routine is much more straightforward than the Fortran 77 version, because Fortran 90 allows recursion. (In fact, this routine is modeled on the C version of `miser`, which was recursive from the start.)

CITED REFERENCES AND FURTHER READING:

- Marsaglia, G., and Zaman, A. 1994, *Computers in Physics*, vol. 8, pp. 117–121. [1]
- Marsaglia, G. 1985, *Linear Algebra and Its Applications*, vol. 67, pp. 147–156. [2]
- Harbison, S.P., and Steele, G.L. 1991, *C: A Reference Manual*, Third Edition, §5.1.1. [3]

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0)
Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software.
Permission is granted for Internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).